Declarative Data Cleaning: Language, Model, and Algorithms

Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, Cristian Saita

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Declarative Data Cleaning: Language, Model, and Algorithms

Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, Cristian Saita

No 4149

March 2001

THÈME 3

ISSN 0249-6399 ISRN INRIA/RR--4149--FR+ENG

INRIA
ROCQUENCOURT

Declarative Data Cleaning: Language, Model, and Algorithms

Helena Galhardas*, Daniela Florescu, Dennis Shasha, Eric Simon, Cristian Saita

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Caravel

Rapport de recherche n ° 4149 — March 2001 — 37 pages

Abstract: The problem of data cleaning, which consists of removing inconsistencies and errors from original data sets, is well known in the area of decision support systems and data warehouses. However, for non-conventional applications, such as the migration of largely unstructured data into structured one, or the integration of heterogeneous scientific data sets in inter-disciplinary fields (e.g., in environmental science), existing ETL (Extraction Transformation Loading) and data cleaning tools for writing data cleaning programs are insufficient. The main challenge with them is the design of a data flow graph that effectively generates clean data, and can perform efficiently on large sets of input data. The difficulty with them comes from (i) a lack of clear separation between the logical specification of data transformations and their physical implementation and (ii) the lack of explanation of cleaning results and user interaction facilities to tune a data cleaning program. This paper addresses these two problems and presents a language, an execution model and algorithms that enable users to express data cleaning specifications declaratively and perform the cleaning efficiently. We use as an example a set of bibliographic references used to construct the Citeseer Web site. The underlying data integration problem is to derive structured and clean textual records so that meaningful queries can be performed. Experimental results report on the assessement of the proposed framework for data cleaning.

Key-words: Data quality, data transformations, similarity join, query optimization

(Résumé : tsvp)

* Founded by "Instituto Superior Técnico" - Technical University of Lisbon and by a JNICT fellowship of Program PRAXIS XXI (Portugal)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Nettoyage de Données Déclaratif: Language, Modèle et Algoritmes

Résumé : Le problème de nettoyage de données qui consiste à éliminer les incohèrences et les erreurs trouvées dans des jeux de données originaux, est bien connu dans le domaine des systèmes d'aide à la décision et des entrepôts de données. Néanmoins, pour des applications non-conventionnelles, telles que la migration de données faiblement structurées vers des données structurées, ou l'intégration de jeux de données scientifiques hetérogènes dans des domaines inter-disciplinaires (e.g., dans les sciences de l'environnement), les outils d'ETL (*Extraction Transformation Loading*) et de nettoyage de données existants sont insuffisants. Leur principal défi est la conception d'un graphe de flots de données qui gènere des données nettoyées d'une manière effective, et qui se comporte de façon efficace en face de grandes volumes d'information. La difficulté sous-jacente est due à: (i) l'absence de séparation claire entre la spécification logique des transformations de données et leur implantation physique; (ii) l'absence d'explication du résultat d'un processus de nettoyage et de modes d'interaction humaine permettant d'affiner un programme de nettoyage de données. Cet article adresse ces deux problèmes et prèsente un langage, un modèle d'execution et des algoritmes qui permettent aux utilisateurs d'exprimer des spécifications de nettoyage de données de façon déclarative aussi bien que d'exécuter le processus de nettoyage efficacement. Nous utilisons comme exemple un ensemble de réferences bibliographiques utilisées auparavant pour contruire le site Web de Citeseer. Le probléme d'intégration de données inhérent est celui de dériver des enregistrements textuels structurés et nettoyés de façon à permettre l'évaluation de requêtes pertinentes. Les résultats expérimentaux présentent l'évaluation de l'environnement de nettoyage de données proposé.

Mots-clé : Qualité de données, transformations de données, jointure approximée, optimisation de requêtes

1 Introduction

The development of Internet services often requires the integration of heterogeneous sources of data. Often the sources are unstructured whereas the intended service requires structured data. The main challenge is to provide consistent and error-free data (aka clean data).

To illustrate the difficulty of data cleaning for the Web, we first introduce a concrete running example. The Citeseer Web site (see [16]) collects all the bibliographic references in Computer Science that appear in documents (reports, publications, etc) available on the Web in the form of postscript, or pdf files. Using these data, Citeseer enables Web clients to browse through citations in order to find out for instance, how many times a given paper is referenced. The data used to construct the Citeseer site is a large set of string records. The next two records belong to this data set:

```
[QGMW96] Dallan Quass, Ashish Gupta, Inderphal Singh Mumick, and Jennifer
Widom. Making Views Self-Maintainable for Data Warehousing. In Proceedings
of the Conference on Parallel and Distributed Information Systems.
Miami Beach, Florida, USA, 1996. Available via WWW at www-db.stanford.edu
as pub/papers/self-maint.ps.

[12] D. Quass, A. Gupta, I. Mumick, and J. Widom: Making views
self-maintanable for data, PDIS'95
```

Establishing that these are the same paper is a challenge. First, there is no universal record key that could establish their identity. Second, there are several syntactic and formatting differences between the records. Authors are written in different formats (e.g. "Dallan Quass" and "D. Quass"), and the name of the conference appears abbreviated ("PDIS") or in full text ("Conference on Parallel ..."). Third, data can be inconsistent, such as years of publication ("1996" and "1995"). Fourth, data can be erroneous due to misspelling or errors introducing during the automatic processing of postscript or pdf files, as in the title of the second record ("maintanable" instead of "maintainable"). Finally, records may hold different information, e.g., city and country are missing in the second record.

1.1 Existing technology

The problem of data cleaning is well known for decision support systems and data warehouses (see [4]). Extraction, Transformation and Loading (ETL) tools and data reengineering tools provide powerful software platforms to implement a large data transformation chain, which can extract data flows from arbitrary data sources and progressively combine these flows through a variety of data transformation operations until clean and appropriately formatted data flows are obtained [23]. The resulting data can then be loaded into some database. Some tools are comprehensive but offer only limited cleaning functionality (e.g., Sagent [26], ETI [17], Informatica [15]), while others are dedicated to data cleaning (e.g., Integrity [28]).

When an application domain is well understood (e.g., cleaning U.S. names and addresses in a file of customers), there exists enough accumulated know-how to guide the design and implementation of a data cleaning program. Thus, designers know which data transformation steps to follow, the operators to use and how to use them (e.g., adjusting parameters). However, for non-conventional applications, such as the migration of largely unstructured data into structured data, or the integration of heterogeneous scientific data sets in cross disciplinary areas (e.g., environmental science), existing data cleaning tools for writing data cleaning programs are insufficient. The main challenge is the design of a data flow graph that effectively generates clean data, and can perform efficiently on large sets of input data. The difficulty comes from (i) a lack of clear separation between the logical specification of data transformations and their physical implementation, and (ii) the lack of data lineage and user interactions facilities to tune a data cleaning program. Relying on our experience with four different commercial ETL tools on the Citeseer data, we argue about each point.

Consider the problem of removing duplicates from the set of author names extracted from the sample Citeseer data set. This problem a priori requires comparing all pairs of author names using a distance function on strings, and producing groups of elements such that all elements of the same group are most probably denoting the same person (e.g., "Dallan Quass" and "D. Quass"). After that, a representative for each group can be chosen (e.g., "Dallan Quass"). Thus, three distinct actions are involved: (1) a *matching* action compares pairs of elements using a distance function; (2) a *clustering* action produces groups of elements; and (3) a *merging* action collapses each group into a single element.

Some tools (e.g., Sagent) do not provide facilities to support the specification and implementation of these actions[1]. Other tools (e.g., Integrity or Informix DataCleanser DataBlade module[8]) provide operators for matching, clustering, and merging. However, in these tools each operator consists of a specific optimized algorithm that enforces a particular semantics for the operation. For instance, in the case of a matching operation between two sets of records, the operator requires that a criteria be satisfied to determine which pairs of records should be compared. In other words, an exhaustive comparison of all pairs of records is not possible, the rationale being that for very large data sets an exhaustive comparison takes too long. Sometimes, the algorithm implemented by the operator is well documented (e.g., multi-pass neighborhood method [13]). In other cases it is either obscure or confidential. This is a problem because the choice of the pairs of records to compare strongly influences the reliability of the result of the matching operation. For instance, it is important to know if successful matches have been lost by the algorithm, when in fact this depends on the mathematical properties of the distance function used in the matching, and the data manipulated. Even if a matching algorithm is documented, this is not satisfactory because a single non exhaustive matching algorithm cannot fit all situations.

To understand the second difficulty, it is important to realize that the more "dirty" the data, the more difficult it is to automate their cleaning with a fixed set of transformations. In the Citeseer example, when the years of publication are different in two records that

[1] Many vendors of ETL tools merely advise their customers to implement these actions using an ad-hoc data processing program in their favorite language or subcontract this development to their consultants.

apparently refer to the same publication, there is no obvious criteria to decide which date to use; hence the user must be explicitly consulted. In existing tools, there is no specific support for user consultation except to write the data to a specific file to be later analyzed by the user. In this case, the integration of that data, after correction, into the data cleaning program is not properly handled. Finally, in existing tools, the process of data cleaning is unidirectional in the sense that once the operators are executed, the only way to analyze what was done is to inspect log files. This is an impediment to the stepwise refinement of a data cleaning program.

1.2 Contributions

This paper describes a data cleaning *framework* that attempts to separate the logical and physical levels. The logical level supports the design of the data flow graph that specifies the data transformations needed to clean the data, while the physical level supports the implementation of the data transformations and their optimization. For instance, at the logical level, a matching operator is provided to assess the similarity of all combinations of records taken from some input data flows, while at the physical level, a specific implementation can be chosen that avoids, say, the evaluation of all record combinations without loosing pertinent combinations. An analogy can be drawn with database application programming where database queries can be specified at a logical level and their implementation can be optimized afterwards without changing the queries.

The features we will emphasize are:

- A declarative language for data cleaning, based on five logical data transformation operators. Each operator can make use of external functions.
- A declarative specification of user interaction based on exceptions that are automatically generated by the execution of operators. A data lineage facility enables the user to interactively inspect intermediate data and exceptions, backtrack the cleaning process, investigate the result of each operation, and modify the data manually.
- A declarative way to specify some properties of distance functions within a matching operation. These properties enable the system to select an optimized implementation for the matching operation.

We developed a system that implements these principles [11]. The paper is organized as follows. Section 2 gives an overview of our data cleaning framework. Section 3 presents the syntax and semantics of our declarative language for specifying a data cleaning program at the logical level. The fourth section explains the execution of data cleaning programs, in particular the optimization problem of the matching operation. Section 5 reports the experiments done and lessons learned using a Citeseer data set of 500,000 records. Section 6 summarizes other related work. Finally, section 7 concludes.

2 Overview of the Data Cleaning Framework

The development of a data cleaning program actually involves two activities. One is the design of the graph of data transformations that should be applied to the input dirty data. The focus there is to define "quality" heuristics that can achieve the best accuracy (i.e., level of cleaning quality) of the results. A second activity is the design of "performance" heuristics that can improve the execution speed of data transformations without sacrificing accuracy. Our framework separates these two activities by providing a logical level where a graph of data transformations is specified using a declarative language, and a physical level where specific optimized algorithms can be selected to implement the transformations.

2.1 Logical Level

To illustrate our approach, suppose we wish to migrate the Citeseer data set (which is a set of strings corresponding to textual bibliographic references) into four sets of structured and clean data, modeled as database relations: *Authors*, identified by a key and a name; *Events*, identified by a key and a name; *Publications*, identified by a key, a title, a year, an event key, a volume, etc; and the correspondence between publications and authors, *Publications-Authors*, identified by a publication key and an author key.

A partial and high-level view of the data cleaning strategy that we used is the following:

1. Add a key to every input record.
2. Extract from each input record, and output into four different flows the information relative to: names of authors, titles of publications, names of events and the association between titles and authors.
3. Extract from each input record, and output into a publication data flow the information relative to the volume, number, contry, city, pages, year and url of each publication.
4. Eliminate duplicates from the flows of author names, titles and events.
5. Aggregate the duplicate-free flow of titles with the flow of publications.

At the logical level, the main constituent of a data cleaning program is the specification of a data flow graph where nodes are data cleaning operations of the following types: mapping, view, matching, clustering, and merging, and the input and output data flows of operators are logically modeled as database relations. Each operator can make use of externally defined functions that implement domain-specific treatments such as the normalization of strings, the extraction of substrings from a string, the computation of the distance between two values, etc.

The design of these operators was driven by two main considerations: (1) *expressiveness*, i.e., the ability to express a large class of data mappings from a set of input relations to another set of output relations, and (2) *usability*, i.e., to provide important data cleaning operations for which specific optimized algorithms have been devised. More specifically, the mapping operator takes a single relation and produces several relations as output; it

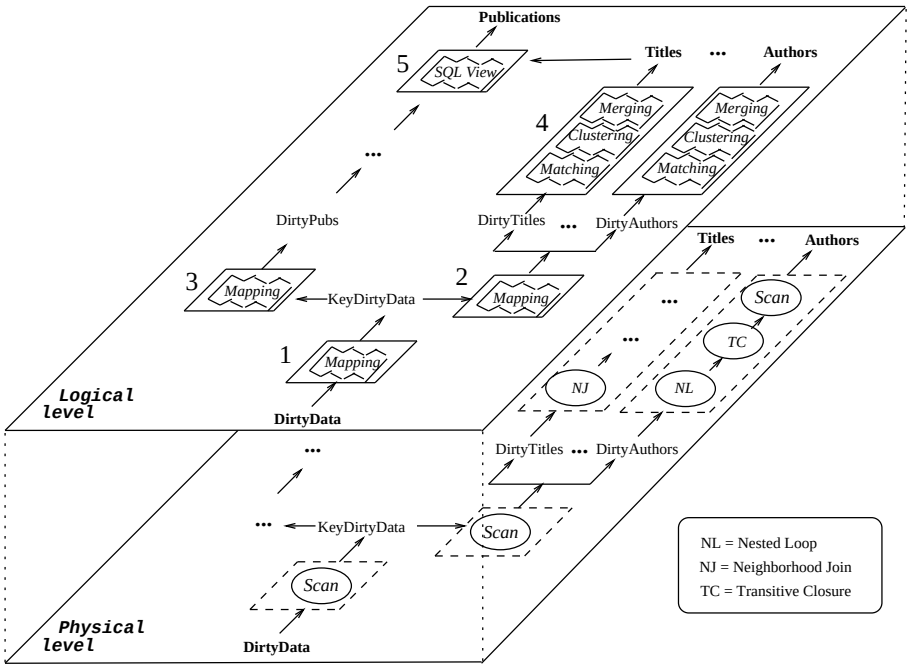

Figure 1: Framework for the bibliographic references

can express any one-to-many data mapping due to its use of external functions. The view operator, which essentially corresponds to an SQL select statement, takes several relations as input and returns a single relation as output; it can express limited many-to-one mappings. The matching operator takes two input relations and produces a single output relation; it computes the similarity between any two input records using an arbitrary distance function. This operation is obviously expressible using a view operator but having it as a first-class operator considerably facilitates its optimization. The clustering operator transforms an input relation into a nested relation where each nested set is a cluster of records from the input relation; the clustering of records is arbitrary. Finally, the merging operator takes a relation representing a set of clusters and apply an arbitrary data mapping to the elements of each cluster.

In addition, our framework is extensible: it allows the definition of new operators from the five basic ones. For instance, [25] presents a method for outlier detection defined as: "A point p in a data set is an *outlier* with respect to parameters k and d if no more than k points in the data set are at a distance of d or less from p". Supposing that the data set is a relation, this is easily expressed in our framework as a sequence of a matching operation using distance d, and a view operation that groups every point with its neighbors and selects all groups with cardinality less than k. Similarly, a *record-linkage procedure* (see e.g., [1]) can be expressed as a sequence of matching, clustering and merging operations.

Example 2.1: The above data cleaning strategy is mapped into the data flow graph of Figure 1. The numbering beside each data cleaning operation corresponds to a step in the strategy. For each output data flow of Step 2, the duplicate elimination is mapped into a sequence of three operations of matching, clustering, and merging. Every other step is mapped into a single operator.

Besides the declarative specification of the data flow graph, a data cleaning program specified by our framework includes sections for declaring the set of constants and data types used throughout the program, the signatures of the externally-defined functions, and the schema of the input data flows of the program expressed as relational schemas.

Our data cleaning system makes two original contributions to simplify the logical design of data cleaning programs. This claim will be substantiated by the report of experiments in Section 5.

- First, a small number of operators having a well defined semantics facilitates the formulation of the data cleaning strategy as a data flow graph[2]. Once a strategy is defined, the designer can focus on refining the cleaning criteria specified by each operator.
- Second, the semantics of each operator includes the automatic generation of a variety of exceptions. At any stage of execution of a data cleaning program, a data lineage mechanism enables users to browse into exceptions, analyze their provenance in the data flow graph and interactively correct the data. Corrected data can then be re-integrated into the data flow graph. This functionality proved essential in our experiments with Citeseer.

2.2 Physical level

At the physical level, certain decisions can be made to speed up the execution of data cleaning programs. First, the implementation of the externally defined functions can be optimized. Second, an efficient algorithm can be selected to implement an operation among a set of alternative algorithms. As explained earlier, a very sensitive operator to the choice of execution algorithm is matching. An original contribution of our data cleaning system is to associate with each optimized matching algorithm, the mathematical properties that the distance function used in the matching operator must have in order to enable the optimization, and the parameters that are necessary to run the optimized algorithm. Then, our system enables the user to specify, within the logical specification of a given matching operator, the properties of the distance function, together with the required parameters for optimization. The system can consume this information to choose an algorithm to implement a matching. The important point here is that users control the proper usage of

[2]By contrast, existing data cleaning tools provide a very large number of kinds of data transformations (e.g., more than 100 in Sagent), ranging from general ones (e.g., arbitrary SQL statement) to very specific-domain ones (e.g., normalization of postal addresses). In addition, as said earlier, several operations, such as matching, lack of a well defined semantics.

optimization algorithms. They first determine (in the logical specification) the matching criteria that would provide accurate results, and then provide the necessary information to enable optimized executions[3]. Figure 1 shows the algorithms selected to implement each logical operation.

3 Data Cleaning Specification Language

This section gives a formal definition of the five logical operators (mapping, matching, clustering, merging and view) offered by our declarative language for expressing data cleaning transformations. These formalities are necessary to give a precise semantics of the exceptions generated by the operators and explain how user-defined external functions take place when executing operators. For each operator, we first give an intuition of its syntax and semantics, then we formally describe its semantics, and finally present a detailed syntax which is in the spirit of SQL. It is important to keep in mind that the data structures and functions used along this section to formally describe the semantics of the operators do not define how operators are actually implemented. Implementation issues are discussed in Section 4.

3.1 Preliminaries

We assume the existence of several pairwise disjoint sets of symbols. First, there is a set of *constants* **dom** called the underlying domain and a set of *attributes* noted **att**. The domain **dom** includes an *exception value*, noted exc, and a null value, noted null. Second, there is a set of *variables* **var** that will be used to range over elements of **dom**; $Dom(A)$ is a set called the *domain* of attribute A. Third, there is a set of n-ary predicate symbols noted **pred**. Each predicate P is associated with a *schema*, noted $sch(P)$, which is a finite set of attributes. A predicate with its schema is called a *relation*[4].

A fact over a relation $P\{A_1, ..., A_n\}$ is an expression of the form $P(a_1, ..., a_n)$, where each a_i is a constant in $Dom(A_i)$. An instance over P is a finite set of facts over P. The instance of the set of all predicates is a finite set of facts called a *database instance*. If I is a database instance, then $I[P]$ is the set of facts over P in I. We assume that each predicate admits one distinguished instance, noted EXC, called *exception instance*.

Finally, there are two sets of externally defined, loop-free and functional n-ary functions: **atom_f** is a set of mono-valued functions that return values in **dom**; and **table_f** is a set of functions that return a set of facts over some predicate. A *functional expression* is an expression $f(t_1, ..., t_n)$, where f is a function, and each t_i is either an element of **dom**, **var**, **pred**, or a functional expression.

[3]By default, a naive execution is always possible

[4]We sometimes indifferently use the terms relation and predicate.

3.2 Mapping Operator

A mapping operator restructures an input relation into possibly many output relations. It can express any one-to-many mapping over an input relation.

Example 3.1: The following mapping operator transforms the relation DirtyData{paper} into a "target" relation KeyDirtyData{paperkey, paper}; this corresponds to Step 1 of Example 2.1. The LET clause contains a statement that constructs a predicate Key using an external (atomic) function generateKey that takes as argument a variable DirtyData.paper ranging over attribute paper of DirtyData. Relation Key is constructed as follows. For every fact DirtyData(a) in the instance of DirtyData[5], if generateKey(a) does not return an exception value exc, then a fact Key(a, generateKey(a)) is added to the instance of Key. Otherwise, a fact DirtyData^{exc}(a) is added to the instance of DirtyData^{exc}. We shall say that this statement "defines" a relation Key{paper, generateKey}[6]. The schema of the target relation is specified by the "{ SELECT key.generateKey AS ...}" clause. It indicates that the schema of KeyDirtyData is built using the attributes of Key and DirtyData. Finally, the constraint stipulates that a paper attribute value must never be null.

```
CREATE MAPPING AddKeytoDirtyData
FROM DirtyData
LET Key = generateKey(DirtyData.paper)
{ SELECT Key.generateKey AS paperKey, DirtyData.paper AS paper INTO KeyDirtyData
CONSTRAINT NOT NULL paper}
```

3.2.1 Semantics of the mapping operator

We shall successively define all the components of the specification of a mapping operator. A main constituent of a mapping operation is the "let-clause" that defines the instances of some predicates that are local to the mapping operation by means of externally-defined functions. These local predicates are used to construct the instances of the target relations.

Definition 3.1: (assignment statement). Let $S\{A_1, ..., A_n\}$ be a relation. Let $\vec{x}$ be a vector of variables $x_1, ..., x_n$ ranging over $Dom(A_1), ..., Dom(A_n)$ respectively.

- An *atomic assignment statement* wrt S is an expression of the form: $P = f(\vec{x}, ...)$ where f is a function of **atom_f** that takes $\vec{x}$ as argument among other possible arguments, and P is a predicate of arity $n + 1$. This statement is said to *define* a predicate P such that $sch(P) = \{A_1, ..., A_n, f\}$.
- A *table assignment statement* wrt S is an expression of the form: $P = f(\vec{x}, ...)$ where f is a function of **table_f** that returns a set of facts of arity k, and P is a predicate of arity $n + k$. If f returns a set of facts over a predicate whose schema is $\{B_1, ..., B_k\}$ then the assignment statement is said to *define* a predicate P such that $sch(P) = \{A_1, ..., A_n, B_1, ..., B_k\}$.

[5] Where a is a string representing a paper.

[6] For convenience, we shall assume that the name of the attribute holding the result of the function is the same as the name of the function.

Example 3.2: The statement Key = generateKey(DirtyData.paper) in the previous example is an atomic assignment statement. Now, consider the following table assignment statements wrt DirtyData:

AuthorTitleEvent = extractAuthorTitleEvent(DirtyData.paper)
AuthorTitleEvent = extractAuthorTitleEventWithDictionary(DirtyData.paper, dictionary)

in which DirtyData.paper is as before, and extractAuthorTitleEvent and extractAuthorTitleEventWithDictionary are table functions that return a relation with schema {authorlist, title, event}. These statements define a relation AuthorTitleEvent{paper, authorlist, title, event}. dictionary is a relation containing a dictionary of names that is used as an argument of the table function.

Definition 3.2: (Let-clause). Let $S\{A_1, ..., A_n\}$ be a relation. A *let-clause* wrt S is a sequence of assignment statements $e_1...e_q$. Assuming that each e_i is of the form $P_i = f_i(\vec{x}, \vec{y_i})$, where $\vec{x}$ is a vector of variables $x_1, ..., x_n$ ranging over $Dom(A_1), ..., Dom(A_n)$ and $\vec{y_i}$ is a vector of additional arguments (e.g., as dictionary in the example above), we shall say that the let-clause *defines* predicates $P_1, ..., P_q$ wrt S. There is the restriction that if t is an element of vector $\vec{y_i}$ in $f_i(\vec{x}, \vec{y_i})$, and t is either a variable ranging over an attribute domain of P_j, or $t = P_j$, then $j > i$.

We now give the semantics of a let-clause. The assignment statements of a let-clause are evaluated in their order of appearance in the let-clause. The evaluation of an assignment $P = f(\vec{x})$[7] wrt S in a database instance I yields a new database instance J, in which the instances of predicates P and S^{exc} (hence, called the *exceptions* of S) are as follows. If f is an atomic function, then for each fact $S(a_1, ..., a_n)$ in $I[S]$, if $f(a_1, ..., a_n)$ returns a value b that is not an exception value exc then a fact $P(a_1, ..., a_n, b)$ is added to $J[P]$. Otherwise the fact $(a_1, ..., a_n)$ that caused the exception is added to $J[S^{exc}]$. Similarly, if f is a table function, then if $f(a_1, ..., a_n)$ returns a set of facts that is not an exception instance EXC, then for each such fact, say $(b_1, ..., b_k)$, a fact $P(a_1, ..., a_n, b_1, ..., b_k)$ is added to $J[P]$, otherwise the fact that caused the exception is added to $J[S^{exc}]$. We shall note Let(E, I) the function that maps a database instance I into a new instance using a let-clause E. A formal definition is given in Appendix 1.

Example 3.3: The evaluation of the table assignment statement of Example 3.2 implies that for each fact DirtyData(a), if extractAuthorTitleEvent(a) returns a set of facts, then for each of these facts, say $(b_1, ..., b_k)$, a fact AuthorTitleEvent(a, b_1, ..., b_k) is added to the instance of AuthorTitleEvent. Otherwise, if extractAuthorTitleEvent(a) returns an exception instance EXC, then a fact DirtyData^{exc}(a) is generated.

Definition 3.3: (Filter). Let $P_1, ..., P_q$ be some predicates. A *filter* over $P_1, ..., P_q$ is a logical condition $\Phi(x_1, ..., x_m)$, where the x_i range over the domains of attributes of $P_1, ..., P_q$.

[7] In the following, in order to simplify notations, we shall often note $f(\vec{x})$ a function that admits $\vec{x}$ as argument among other possible arguments.

We shall say that Φ *defines* a predicate U whose schema is the union of the schemas of $P_1, ..., P_q$.

Intuitively, a filter over $P_1, ..., P_q$ constructs the instance of a predicate U by performing a Cartesian product of all the instances of $P_1, ..., P_q$ and then applying a selection that eliminates all the facts that do not satisfy condition Φ. Thus, the schema of U is always implicitly defined as the union of the schemas of $P_1, ..., P_q$. We shall note $\mathsf{filter}(\Phi, I)$ the function that maps a database instance I into a new instance using the filter Φ.

A formal definition is given in Appendix 1.

Example 3.4: A filter over AuthorTitleEvent and Key, as defined in Example 3.2, could be a logical condition as:

AuthorTitleEvent.title $\neq$ null and length(Key.paper) $>$ 10
and Key.paper = AuthorTitleEvent.paper

The evaluation of this filter returns an instance of a relation U {generateKey, paper, author, title, event} that satisfies the logical condition.

Definition 3.4: (Constraint-clause). Let $S\{A_1, ..., A_n\}$ be a relation and $P\{B_1, ..., B_m\}$ be a relation whose schema includes attributes $A_1, ..., A_n$. Then a *constraint* over P wrt $\{A_1, ..., A_n\}$, is a logical condition, noted $\sigma(\vec{x})$, where $\vec{x}$ is a vector of variables $x_1, ...x_m$ ranging over $Dom(B_1)$, ..., $Dom(B_m)$. A set of constraints over P wrt $\{A_1, ..., A_n\}$ forms a *constraint-clause* that *defines* a predicate P^{ok} whose schema is the same as P.

A set Σ of constraints over $P\{B_1, ..., B_m\}$ wrt $\{A_1, ..., A_n\}$ has the following meaning. For every fact $P(b_1, ..., b_m)$ over P, all the constraints $\sigma(b_1, ..., b_m)$ of Σ are evaluated. If no constraint is violated then a fact $P^{ok}(b_1, ..., b_m)$ over some predicate P^{ok} is generated, otherwise the fact obtained by projecting out $P(b_1, ..., b_m)$ on attributes $A_1, ..., A_n$ is added to $I[S^{exc}]$. We shall note $\mathsf{check}(\Sigma, I)$, the function that maps a database instance I into a new instance using a set Σ.

Example 3.5: A constraint over a relation U {generateKey, paper, author, title, event} wrt {paper} could be a logical condition $\sigma(\vec{x})$ that evaluates to true if there is no fact other than $\vec{x}$ in U that has the same title, author, and event. Its evaluation in instance I works as follows. For each fact $\mathsf{U}(\vec{b})$, if the constraint is satisfied then a new fact $\mathsf{U}^{\mathsf{ok}}(\vec{b})$ is added to the instance of U^{ok}\{generateKey, paper, author, title, event\}, otherwise a fact corresponding to the projection of $\vec{b}$ over attribute paper is added to the instance of $\mathsf{DirtyData}^{exc}$\{paper\}.

A formal definition of the semantics of a constraint-clause is given in Appendix 1.

Definition 3.5: (Mapping operator). A mapping operator is a 5-tuple $(S, E, \Phi, \mathcal{T}, \mathcal{C})$, where $S\{A_1, ..., A_n\}$ is a relation; E is a let-clause wrt S that defines predicates $P_1, ..., P_q$;

Φ is a filter over $P_1, ..., P_q$; $\mathcal{T}$ is a set of predicates $T_1, ..., T_m$, called *target* predicates, where $sch(T_i) = \{B_{i_1}, ..., B_{i_k}\}$ and each B_{i_j} is an attribute of the schema of S, or P_1, ..., or P_q; and $\mathcal{C}$ is a set $\{\mathcal{C}_1,...,\mathcal{C}_m\}$ where each $\mathcal{C}_i$ is a set of logical conditions over T_i.

The semantics of a mapping operator is now given. First, the let-clause is evaluated to generate the instances of predicates $P_1, ..., P_q$. This may generate exceptions in the instance of S^{exc}. Note that by definition of the let-clause, the schema of all the P_i contains all the attributes of S. Then the filter is evaluated to generate an instance of a predicate U by doing a cartesian product of the instances of $P_1, ..., P_q$, and eliminating all the facts of U that do not satisfy Φ. Next, for each "target" relation $T_i\{B_{i1}, ..., B_{ik}\}$, the relation $U\{A_1, ...A_n, ...\}$ is projected on attributes $\{A_1, ..., A_n, B_{i1}, ..., B_{ik}\}$ to generate a relation $pre\_T_i\{A_1, ..., A_n, B_{i1}, ..., B_{ik}\}$. Notice that, by definition of the mapping operator, each attribute of the schema of T_i is also an attribute of the schema of U. Finally, each set of conditions $\mathcal{C}_i$ yields a constraint-clause Σ_i over pre_T_i wrt S. Each Σ_i is evaluated and produces an instance of $pre\_T_i^{ok}\{A_1, ..., A_n, B_{i1}, ..., B_{ik}\}$ that is projected on the attributes of T_i to produce the resulting instance of the T_i. Here again, exceptions can be generated in the instance of S^{exc} during the phase of constraint checking. The diagram of Figure 2 summarizes the sequence of database mappings used to define the semantics of the mapping operator.

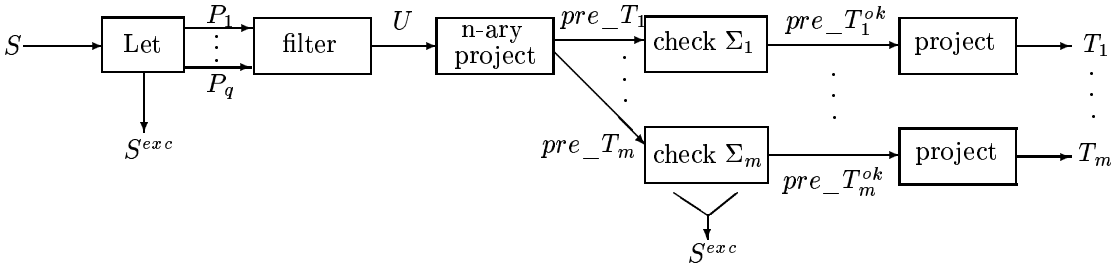

Figure 2: Semantics of the mapping operator.

3.2.2 Syntax of the create mapping clause

We now present the SQL-like syntax for the mapping operator, whose general structure is given in Table 1. A BNF-like syntax is used to describe the language statements. Within this grammar, non-terminal symbols are enclosed in angle brackets <>; terminal symbols that are keywords of the language are in boldface; alternative productions are introduced with |; $[a]$ means that a is optional; and $\{a\}$ ... means that a is repeated one or more times.

The **create** clause indicates the name of the operation. The **from** clause is a standard SQL from-clause that specifies the name of the input predicate of the mapping operator.

Table 1: Syntax for a mapping operator

<mapping-operator>	:	**create mapping** <operation-name> **from** <predicate-name> [<alias-variable>] [**let** <let-clause>] [**where** <where-clause>] <output-clause>

Then the **let** keyword introduces a let-clause as defined before. The syntax of an assignment statement is given in Table 2. A predicate can be assigned a functional expression directly as in Definition 3.1. However, as a syntactic sugaring, the syntax also allows to assign a predicate using an **if then else** control structure, in order to expose within the operator the logic of the assignment. Furthermore, in the case of a table assignment statement, a predicate can be assigned the result of an SQL **select from where** expression that can make use of previously defined predicates. There is the possibility to explicitly throw an exception, introduced by the **throw** keyword, in an if-then-else expression. The syntax also provides two simplifications in the notations. First, in a function-assignment, since the variables ranging over all the attributes of the input relation are always implicit, they can be omitted. Thus, in Example 3.1, we can write Key = generateKey() and omit variable DirtyData.paper. Second, when P is defined using an atomic function foo, we abusively allow to use expression P as a shorthand for P.foo. The **where** keyword introduces a filter expressed as a conjunctive normal form in a syntax similar to an SQL where clause[8].

Table 2: Syntax for a let-clause

<let-clause>	:	<assignment-statement> , [{<assignment-statement>} ...]
<assignment-statement>	:	<predicate-name> = <functional-assignment>
<functional-assignment>	:	<functional-expression> | <if-then-else-expression> | <sfw-expression>
<if-then-else-expression>	:	**if** <condition> **then** <thenelse-expression> [**else** <thenelse-expression>]
<thenelse-expression>	:	**throw** <exception-name> | <functional-assignment>
<functional-expression>]	:	<function-name>(<arg-expression> [{,<arg-expression>} ...])
<arg-expression>	:	<constant> | <domain-variable> | <predicate-name> | <functional-expression>
<sfw-expression>	:	**select** <sql-project-clause> **from** <sql-from-clause> **where** <sql-where-clause>

The syntax of the output-clause is given in Table 3. It consists of a sequence of **select into** expressions that specify the schema of each target relation, and the constraints associated with each target relation. Constraints can be of the following kinds: **not null**, **unique**, **foreign key** and **check**. Their syntax is the same as SQL assertions, but their meaning is

[8] Implicit join conditions on attributes of $P_1, ..., P_q$, as Key.paper = AuthorTitleEvent.paper in Example 3.4, are always omitted.

different due to the management of exceptions when constraints are violated (the operation does not stop), as explained before.

Table 3: Syntax for an output-clause

<output-clause>	:	<select-into-clause> [{<select-into-clause>} ...]
<select-into-clause>	:	**{** <sql select-into> [{**constraint** <constraint-clause>} ...] **}**
<sql select-into>	:	**select** <sql-project-clause> **into** <predicate-name>
<constraint-clause>	:	**unique** <att-name> [{<att-name>} ...] | **not null** <att-name> [{<att-name>} ...] | **foreign key** <att-name> [{<att-name>} ...] **references** <predicate-name> (<att-name> [{<att-name>} ...]) | **check** <where-clause>

Example 3.6: The mapping command below transforms KeyDirtyData defined in Example 3.1 into four target relations, whose schemas are specified elsewhere in the sections of the data cleaning program that declare the externally defined functions. The schemas of the predicates returned by table functions extractAuthorTitleEvent and extractAuthors are {authorlist, title, event} and {id, name} respectively. In the case of a table assignment statement, a predicate can be assigned the result of an SQL **select from where** expression that can make use of previously defined predicates.

```
CREATE MAPPING Extraction
FROM KeyDirtyData kdd
LET AuthorTitleEvent = extractAuthorTitleEvent(kdd.paper),
    AuthId = SELECT id, name
             FROM extractAuthors(AuthTitleEvent.authorlist)
WHERE AuthorTitleEvent.title <> null and length(kdd.paper) > 10
{ SELECT kdd.paperKey AS pubKey, AuthorTitleEvent.title AS title, kdd.paperKey AS eventKey INTO DirtyTitles
CONSTRAINT NOT NULL title}
{ SELECT kdd.paperKey AS eventKey, AuthorTitleEvent.event AS event INTO DirtyEvents
CONSTRAINT NOT NULL event}
{ SELECT AuthId.id AS authorKey, AuthId.name AS name INTO DirtyAuthors
CONSTRAINT NOT NULL name }
{ SELECT AuthId.id AS authorKey, kdd.paperKey AS pubKey INTO DirtyTitlesDirtyAuthors}
```

Suppose that the instance of KeyDirtyData has the following two facts:

```
KeyDirtyData: 25 | [12] D. Quass, A. Gupta, I. Mumick, and J. Widom: Making views self-maintanable for data,
              PDIS'95
              26 | [QGMW96] Dallan Quass, Ashish Gupta, Inderphal Singh Mumick, and Jennifer Widom.
              Making Views Self-Maintainable for Data Warehousing. In Proceedings of the
              Conference on Parallel and Distributed Information Systems. Miami Beach, Florida,
              USA, 1996. Available via WWW at www-db.stanford.edu as pub/papers/self-maint.ps
```

Then the instances of the four target predicates contain the following facts:

```
DirtyTitles: 25 | Making views self-maintanable for data | 25
             26 | Making Views Self-Maintainable for Data Warehousing | 26

DirtyEvents: 25 | PDIS
             26 | Proceedings of the Conference on Parallel and Distributed Information Systems

DirtyAuthors: 1 | D. Quass
              2 | A. Gupta
              6 | Dallan Quass...

DirtyTitlesDirtyAuthors: 1 | 25
                         2 | 25
                         6 | 26 ...
```

3.3 Matching Operator

A matching operator computes a distance value for each pair of facts in the Cartesian product of two input relations. The following example illustrates this.

Example 3.7: This (self-)matching operator takes as input the relation DirtyAuthors{authorKey, name} twice. Its intention is to find possible duplicates within DirtyAuthors. The LET clause has the same meaning as before with the additional constraint that it *must* define a predicate, named distance, within an atomic assignment statement. Here, distance is defined using an atomic function editDistanceAuthors computing an integer distance value between two author names. The LET clause produces a relation distance{authorKey1, name1, authorKey2, name2, editDistanceAuthors} whose instance has one fact for every possible pair of facts taken from the instance of DirtyAuthors. The WHERE clause filters out the facts of distance for which editDistanceAuthors returned a value greater than a value computed (by maxDist) as 15% of the maximal length of the names compared. Finally, the INTO clause specifies the name of the target relation (here, MatchAuthors) whose schema is the same as distance.

```
CREATE MATCHING MatchDirtyAuthors
FROM DirtyAuthors a1, DirtyAuthors a2
LET distance = editDistanceAuthors(a1.name, a2.name)
WHERE distance < maxDist(a1.name, a2.name, 15)
INTO MatchAuthors
```

3.3.1 Semantics of the matching operator

The semantics of the matching operator is based on the notions of let-clause, filter and project.

Definition 3.6: (Matching operator). A matching operator is a 5-tuple $(\mathcal{S}, E, \Phi, \mathcal{S}^{no-match}, T)$, in which $\mathcal{S}$ is a set of predicates $\{S_1, S_2\}$; E is a let-clause wrt the set of all the attributes of S_1, S_2 that defines predicates $P_1, ..., P_q$, and one distinguished predicate $Dist$ such that $Dist$ is assigned in an atomic assignment statement using an atomic function; Φ

is a filter over $Dist, P_1, ..., P_q$; $\mathcal{S}^{no-match}$ is a subset of $\mathcal{S}$; and T is a predicate called the *target* predicate of the matching operator.

The semantics of a matching operator is defined as follows. First, a filter $\Phi' = \mathsf{true}$ over S_1, S_2 is evaluated to return a predicate U whose instance consists of the cartesian product of the instances of S_1, S_2. Second, the let-clause E is evaluated wrt U. In particular, this returns an instance of $Dist$ whose schema includes all the attributes of S_1 and S_2. Intuitively, the atomic function used in the assignment of $Dist$ computes a "distance" value (a real number) between 2 facts taken from S_1, S_2. Note that the let-clause may also generate exceptions into the instance of some predicate U^{exc}.

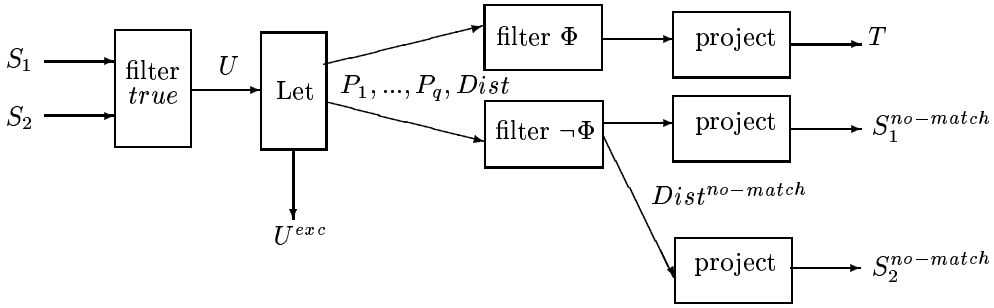

Figure 3: Semantics of the matching operator.

Third, Φ is evaluated and its result is projected on $Dist$ attributes to return an instance of T. Intuitively, the instance of T only contains those pairs of facts (with their distance value), which make sense with regard to condition Φ. Fourth, a filter $\neg\Phi$ is formed and evaluated and its result is projected on $Dist$ attributes to return an instance of a predicate $Dist^{no-match}$. Finally, for each predicate S_i of $\mathcal{S}^{no-match}$, an instance of $S_i^{no-match}$ containing all the facts of $Dist^{no-match}$ projected on their S_i-attributes is generated. The instances of U^{exc}, $S_1^{no-match}$, $S_2^{no-match}$ and T form the output of the matching operator. The diagram of Figure 3 summarizes the sequence of database mappings used to define the semantics of the matching operator.

3.3.2 Syntax of the create matching clause

The general structure of the matching operator is given on Table 4. The syntax for the components of the operator have already been presented before. The only subtlety in the SQL-like syntax of the matching operator is the use of the symbol "+" after an input predicate in the **from** clause (it would be "a1 +" in the above example). It indicates that an instance of a predicate containing all the facts of DirtyAuthors that did not match (called $\mathsf{DirtyAuthors}^{no-match}$ in the formal definition before), must be returned by the operator.

Table 4: Syntax for a matching operator

<matching-operator>	:	**create matching** <operation-name> **from** <predicate-name> [+][<alias-variable>] [{<predicate-name> [+] [<alias-variable>] ...}] [**let** <let-clause>] [**where** <where-clause>] **into** <predicate-name>

3.4 Clustering Operator

A clustering operation applies to a relation generated by a matching operation. As explained before, a matching operation, taking two inputs S_1 and S_2, generates an instance of the target predicate T in which each element consists of a pair of facts over S_1, S_2 together with a distance value. Consider the set $\mathcal{F}$ of all the facts over S_1, S_2 that appear in this instance of T. The clustering operator applies a clustering method to this set and produces a set of clusters, each of which is a subset of $\mathcal{F}$. The clusters can be disjoint or not depending on the clustering method. The following example illustrates this.

Example 3.8: Consider the predicate MatchAuthors generated by the matching operation of Example 3.7. The purpose of a clustering operation over MatchAuthors is to produce a set of clusters, each consisting of a set of DirtyAuthors facts that are sufficiently close to each other and probably correspond to the same author. One possible clustering method is to view each fact of MatchAuthors as a binary relationship between DirtyAuthors facts, and group in the same cluster all facts that are transitively connected. In the formal definition, the result of the clustering operation is a relation that has one attribute, clust_id, and as many attributes as there are input relations to the matching operation that defined MatchAuthors, each of which holds the identifier of a fact in the corresponding input relation[9]. Thus, in our example, the output of the clustering operation over MatchAuthors is formally a relation with three attributes, one for each of the two DirtyAuthors relations, and one clust_id attribute. Suppose that we apply the clustering operation using a transtive closure to the following facts of MatchAuthors:

MatchAuthors: 1 | D Quass | 6 | Dallan Quass | 1
1 | D Quass | 7 | Quass | 1
2 | A Gupta | 10 | H Gupta | 1

Then, assuming that o_1, o_2, o_3, o_4, and o_5 are the identifiers for authors 1, 2, 6, 7, 10, we would have the following facts in the output relation, say clusterAuthors:

clusterAuthors: 1 | o_1 | null
1 | o_3 | null
1 | o_4 | null

[9] In practice, we implemented the result of a clustering operation differently.

1 | null | o_1
1 | null | o_3
2 | o_2 | null
2 | o_5 | null
2 | null | o_2
2 | null | o_5

3.4.1 Semantics of the clustering operator

We first introduce auxiliary definitions and the notion of clustering method. We assume the existence of several countably infinite pairwise distinct sets of symbols such that for each predicate P, there is a set of constants **P_id**, called *identifiers*. We further assume that each fact over P can be mapped to a unique identifier of **P_id** using an injective function.

Given this, suppose that T is the target predicate of a matching operation over predicates S_1 and S_2. We define Identify_T as a predicate of schema $\{Id_1, Id_2, dist\}$, where $Dom(Id_i) =$ **S_i_id** and $Dom(dist)$ is the set of real numbers. If $I[T]$ is an instance of T, then for each fact $T(\vec{a_1}, \vec{a_2}, d)$ of $I[T]$, such that $S_i(\vec{a_i})$ is a fact over S_i, there is a fact (o_1, o_2, d) in I[Identify_T]. Essentially, each fact over Identify_T is a pair of identifiers associated with a distance value.

Definition 3.7: (Clustering method). Let T be the target predicate of a matching operation over predicates S_1 and S_2. Let I be a database instance. Let $\mathcal{O}$ be the set of all identifiers o_i that appear in the facts (o_1, o_2, d) of I[Identify_T]. Then, a clustering method over Identify_T is a function that maps $\mathcal{O}$ into a set of *clusters* $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$, such that for each i, $1 \leq i \leq k$, $\mathcal{O}_i \subseteq \mathcal{O}$.

Many clustering methods exist (see e.g., [18]), each of which provides certain properties to the clusters they produce. For instance, the transitive closure method presented earlier produces disjoint clusters.

Definition 3.8: (Clustering operator). A clustering operator is defined by a 3-tuple (T, clust, T'), in which T is the target predicate of a matching operation over predicates S_1 and S_2; clust is a clustering method over Identify_T; and T' is a predicate, called *target*, of schema $\{clust\_id, Id_1, Id_2\}$, where $Dom(clust\_id)$ is the domain of integers and $Dom(Id_i) =$ **S_i_id**.

The evaluation of a clustering operation in a database instance I is defined as follows. First, the clusters $\{\mathcal{O}_1, ...\mathcal{O}_k\}$ are produced from clust(Identify_T, I). Second, if o is an identifier of **S_i_id** that appears in a cluster $\mathcal{O}_j$, $1 \leq j \leq k$, then there is a fact $T'(j, a_1, a_2)$ in J, where for each k, $k \in \{1, 2\}$, if $k \neq i$ then $a_k =$ null else $a_k = o$.

Example 3.9: In the previous example, if $(o_1, o_3, 1)$ and $(o_1, o_4, 1)$ are facts over Identify_MatchAuthors, then by transitive clustering, they belong to the same cluster.

3.4.2 Syntax of the create clustering clause

The general syntax for a clustering operation is given in Table 5. Some methods require that parameters be passed as arguments in the **with parameters** clause. For instance, if we use a "nearest-neighbor" clustering method, a parameter is the maximum distance from the centroid of the cluster.

Table 5: Syntax for a clustering operator

<clustering-operator>	:	**create clustering** <operation-name> **from** <predicate-name> [<alias-variable>] **by method** <method-name> [**with parameters** <parameter-name> [{<parameter-name>} ...]] **into** <predicate-name>

Example 3.10: The following is a specification of the clustering operation by transitive closure; the schema of the target predicate clusterAuthors is {cluster_id,DirtyAuthors_key1,DirtyAuthors_key2}.

```
CREATE CLUSTERING clusterAuthorsByTranstiveClosure
FROM MatchAuthors
BY METHOD transitive closure
INTO clusterAuthors
```

3.5 Merging Operator

The merging operation applies to the result of a clustering operation, collapsing each cluster into a single fact over the target relation.

Example 3.11: Consider the clusterAuthors predicate obtained in the previous example. Each cluster contains a set of names. For each cluster, a possible merging strategy is to generate a fact composed of a key value, e.g., generated using a generateKey function, and a name obtained by taking the longest author name among all the author names belonging to the same cluster. Thus, the format of the output relation of the merging operation would be a relation, say Authors, of schema {authorKey, name}.

3.5.1 Semantics of the merging operator

Definition 3.9: (Merging operator). A merging operator is defined by a 5-tuple $(T, E, \Phi, T', \mathcal{C})$, in which T of schema $\{clust\_id, Id_1, ..., Id_m\}$ is the target predicate of a clustering operation; E is a let-clause wrt T projected on its $clust\_id$ attribute, that defines predicates $P_1, ..., P_q$; Φ is a filter over $P_1, ..., P_q$; T' is a predicate, called *target*, of schema $\{B_1, ..., B_k\}$,

such that each B_i $1 \leq i \leq k$ is also an attribute of some predicate P_j, $1 \leq j \leq m$; and $\mathcal{C}$ is a set of logical conditions over T'.

The evaluation of a merging operation (T, E, Φ, T') in a database instance is defined as follows. First, T is projected on its *clust_id* attribute, which returns a relation, say T_c. Then, the let-clause E is evaluated to generate the instances of $P_1, ..., P_q$. Second, the filter Φ is evaluated to generate an instance of a predicate U. Third, the set of conditions $\mathcal{C}$ is used to form a constraint-clause Σ over U wrt T_c. The constraint clause is evaluated and produces an instance of U^{ok} that is projected on the B_i attributes to generate the instance of the target predicate T.

The diagram of Figure 4 summarizes the mappings used to describe the semantics of a merging operator.

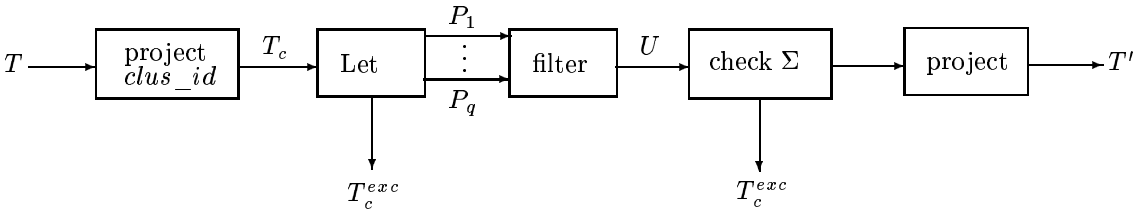

Figure 4: Semantics of the merging operator.

3.5.2 Syntax of the create merging clause

The general syntax for the merging operation is given in Table 6. The **using** clause is similar to a **from** clause with the following difference. A **let** clause is always defined wrt the predicate(s) indicated in the **from**. In the case of merging, as explained before, the **let** clause is defined wrt the clust_id attribute of the predicate indicated in the **using** clause. Essentially, each assignment statement is evaluated by iterating over the clusters of the input relation.

Table 6: Syntax for a merging operator

<merging-operator>	:	**create merging** <operation-name> **using** <predicate-name> [<alias-variable>] **let** <let-clause> [**where** <where-clause>] <select-into-clause>

The **let** clause is used to construct the attribute values that will compose each fact over the target predicate. A specific notation is introduced to ease the access to the attribute

values of the elements of a cluster, which are identifiers. Suppose that the identifier's attributes of the input relation, say P, are associated with relations $S_1, ..., S_m$. Let A be an attribute of S_1. Then, if p is a variable ranging over the attribute domain clust_id of P, the expression S_1(p).A refers to the set of facts: {x.A | x is a fact over S_1 and the identifier of x belongs to cluster p}.

Example 3.12: This specification describes the merging operation introduced intuitively in Example 3.10. In this example, ca is a variable ranging over the clust_id attribute of clusterAuthors. Therefore, expression (DirtyAuthors(ca).name) refers to the set of author names associated with all the DirtyAuthors identifiers of cluster ca. This set is passed to the function getLongestAuthorName.

```
CREATE MERGING MergeAuthors
USING clusterAuthors ca
LET name = getLongestAuthorName(DirtyAuthors(ca).name)
    key = generateKey()
{ SELECT key AS authorKey, name AS name INTO Authors }
```

3.6 View Operator

The last logical operator corresponds to an SQL query, augmented with some integrity checking over its result. Hence, we directly introduce its syntax in Table 7. The interpretation of this clause is first to compute the result of the SQL select statement formed from the **select-into clause**, the **from** clause, and the **where** clause. Then, the set of constraints is evaluated against this result. If a constraint is violated, exceptions are generated.

Table 7: Syntax for a view operator

<merging-operator>	:	**create view** <operation-name> **from** <predicate-name> [<alias-variable>] [{<predicate-name> [<alias-variable>] ...}] [**where** <where-clause>] <select-into-clause>

4 Execution of data cleaning programs

Our data cleaning system takes a specification of a data cleaning program expressed in the declarative language and generates a corresponding Java program, in which each operator's specification is translated into a Java class. Several important optimization decisions are made during the code generation. In this section, we briefly present some techniques that our system used to support these decisions, and then focus on the particular case of matching.

4.1 General techniques

4.1.1 Implementation of input and output data

The relations logically consumed and produced by an operator can be implemented either as flat files or as tables in a relational DBMS. The decision to use one or the other implementation depends on the type of operator, and the kinds of clauses that need to be evaluated within the operator. A view operator is always implemented by an SQL select-from-where[10], in order to take advantage of the query processing capabilities of the underlying relational system; hence its input must be loaded as tables in the relational DBMS. If the let-clause of a mapping operator only consists of atomic assignments, as in Example 3.1, a mapping operator is executed by an SQL statement (or a sequence of SQL statements if there are several targets). Otherwise, as in Example 3.6, the mapping is implemented by some Java code, which experimentally provides better performance. A matching operation logically consists of invoking external functions over each element of the Cartesian product of a list of input relations. Since this kind of processing does not perform efficiently in a relational DBMS, a matching is always implemented as Java code[11]. A clustering operation is always implemented by some Java code because it merely consists of invoking a clustering method typically implemented in Java or C over the result of a matching operation. A merging operation consists of iterating over a set of clusters and applying to each cluster an external function. This kind of processing is not executed efficiently in a relational DBMS because it requires a nested query with a table function in the inner query; hence, it is implemented as Java code.

A unique physical identifier is associated with every fact of the input relation of a logical operator[12]. These identifiers are used in the result of a logical operation (e.g., a mapping or a matching), to avoid propagating all the attribute values of the facts originating from the input relations. This technique is well known in relational databases. In the case of a clustering operator, the target relation is vertically partitioned so that there is one output relation associated with each distinct input relation of the matching operator that precedes clustering. For instance, the clusterAuthors relation presented in Example 3.8 is actually modeled as a relation with two attributes respectively holding cluster identifiers and identifiers of Dirty Authors.

4.1.2 Optimization of the WHERE and LET clauses

A first optimization is to evaluate the part of a where-clause that expresses a condition over the attributes of the input relations before evaluating the let-clause. This applies to the cases of mapping and matching operations. Doing this avoids the evaluation of

[10] All SQL statements are encapsulated in JDBC statements callable from Java.

[11] The execution plan generated is a nested loop with external function calls. Even if a where-clause specifies a condition such as distance < maxDist, it does not help because no index-based optimization is available for external functions.

[12] In Oracle, we used *rowids* as unique identifiers.

assignment statements (and hence, possible expensive computations) over elements of the Cartesian product of the input relations that do not satisfy the where-clause. For instance, in the Extraction mapping of Example 3.6, the let-clause is not evaluated for the tuples of KeyDirtyData whose title attribute value is a null value. A second optimization applies to the case of a matching. In this case, the code generated for the operation eliminates exact duplicates in each input relation before applying the distance function specified in the LET clause.

4.2 Implementation of matching - optimization problem

A matching operator with an acceptance distance of ϵ computes a distance value for every pair of tuples taken from two input relations, and returns those pairs of tuples (henceforth, called *candidate matches*) that are at a maximum distance of ϵ from each other. In fact, since the distance function is an approximation of the actual closeness of two records, a subsequent step must determine which of the candidate matches are the *correct matches* (i.e., the pairs of records that really correspond to the same individual).

For very large data sets, the dominant factor in the cost of a matching is the Cartesian product between the two input relations. There are two main kinds of optimizations that enable to reduce this cost. The first one is to preselect the elements of the Cartesian product for which the distance function must be computed, using a "distance filter" that allows some *false matches* (i.e., pairs of records that are falsely declared to be within an ϵ distance), but no *false dismissals* (i.e., pairs of records falsely declared to be out of an ϵ distance). This pre-selection of elements is expected to be cheap to compute. A second type of optimization is to use an approximate method that compares a limited number of records with a good expected probability that most candidate matches will be returned.

4.3 Distance-filtering optimization

This type of optimization has been successfully used for image retrieval [10]. Formally, the result of a matching between two input relations S_1 and S_2 in which the distance, $dist$, between two elements of S_1 and S_2 is required to be less than some ϵ, is a set:

$$\{(x, y, dist(x, y)) \mid x \in S_1 \wedge y \in S_2 \wedge dist(x, y) \leq \epsilon\} \quad (1)$$

The distance filtering optimization requires finding a mapping f over sets S_1 and S_2, with a distance function $dist'$ much cheaper than $dist$, such that:

$$\forall x, \forall y,\ dist'(f(x), f(y)) \leq dist(x, y) \quad (2)$$

Having determined f and $dist'$, the optimization consists of computing the set of pairs (x, y) such that $dist'(f(x), f(y)) \leq \epsilon$, which is a superset of the desired result:

```
Input: S1, S2, dist, ε, dist', f
{
    let S1 be the smaller of the two sets (in number of bytes)
    and let S2 be the larger of the two sets
    if S1 fits into memory then {
        partition (in memory) the set S1 according to f
        for each s2 ∈ S2, compute d(s1, s2) for each element in the partitions P
            of S1 whose f value, noted fP, is such that dist'(fP, f(s2)) ≤ ε
        }
    else {
        partition S1 according to f
        partition S2 according to f
        for i = smallest f of S1 to largest f of S1 {
            read the partitions P of S2, whose f value is such that
                dist'(i, fP) ≤ ε, and that are not already in memory
            compute dist(s1, s2) for each element s1 in a partition whose
                f value is i, and each element s2 in the above partitions.
        }
}
```

Figure 5: Neighborhood Join algorithm

$$Dist\_Filter = \{(x,y) \mid x \in S_1 \wedge y \in S_2 \wedge dist'((f(x), f(y)) \leq \epsilon\}$$

Given this, the set defined by (1) is equivalent to:

$$\{(x, y, dist(x,y)) \mid (x,y) \in Dist\_Filter \wedge dist(x,y) \leq \epsilon\} \quad (3)$$

A generic algorithm that implements this optimization is shown in Figure 5. This algorithm, called *Neighborhood Join* or NJ for short, is effective when both the number of partitions generated by the mapping f, and the number of elements in the partitions selected by the condition on $dist'$ wrt ϵ, are much smaller than the size of the original input data set.

This optimization is illustrated below on a matching operation of the Citeseer data cleaning program that takes as input the relation DirtyTitles{pubKey, title, eventKey} twice, and is specified as shown below (the line between the %'s is explained later). We assume that maxDist is an integer. The editDistanceTitles function is based on the Damerau-Levenshtein metric [27] that returns the number of insertions, deletions and substitutions needed to transform one string into the other .

Example 4.1:

CREATE MATCHING MatchDirtyTitles

```
FROM DirtyTitles p1, DirtyTitles p2
LET distance = editDistanceTitles(p1.title, p2.title)
WHERE distance < maxDist
%distance-filtering: map=length; dist=abs %
INTO MatchTitles
```

The Damerau-Levenshtein edit-distance function has the property of always returning a distance value bounded by the difference of lengths l of the strings compared. Thus, if l exceeds the maximum allowed distance maxDist, there is no need to compute the edit distance because the two strings are undoubtedly dissimilar. This property suggests using as mapping f the function computing the length of a string, and as $dist'$ a function abs such that $\mathsf{abs}(x, y) = |x - y|$.

A key feature of our data cleaning framework is to enable the specification of the properties of the distance function that can be used to optimize the execution of the matching as annotations in the create matching clause. The above example shows the annotation (between the "%s") for the distance filtering property[13]. Here, the type of property is specified as well as the mapping and distance functions, whose code must be provided to the system. These annotations are used by the data cleaning system to guide the code generation for an operation. In our example, the system has the choice between the naive execution and neighborhood join.

4.4 Approximate methods

A virtue of the distance filtering optimization is that, as a consequence of Equation (2), it does not allow false dismissals. The optimization presented here optimizes the computation of candidate matches at the risk of losing some candidate matches at the end. A good representative of this type of method is the multi-pass neighborhood method (MPN) proposed in [13].

The MPN method consists of repeating the two following steps after performing the outer-union of the input relations: 1) choose a key (consisting of one or several attributes, or substrings within the attributes) for each record and sort records accordingly; 2) compare those records that are close to each other within a fixed, usually small, sized window. The criteria for comparing records is defined by a distance function encoded in a proprietary programming language (for instance, "C") [14]. Each execution of the two previous steps (each time with a different key) produces a set of pairs of matching records (i.e., candidate matches). Formally, suppose that there are n records to be matched and the size of the window is p, $p \leq n$. Then $(n - p) + 1$ windows will be defined over the n sorted records. Each window, noted W_i, contains the list of records from record i to record $i + p$. Thus, for a single pass, given $O_i = \{(x, y, dist(x, y)) \mid x \in W_i, y \in W_i, dist(x, y) \leq \epsilon\}$, the result of

[13]This obviously supposes that when the NJ algorithm was registered within the data cleaning system, the parameters map and dist required by the algorithm were declared to the system.

MPN is:

$$\bigcup_{i=1}^{(n-p)+1} O_i$$

When there are several passes, a final transitive closure is applied to all the pairs of records that have been returned, yielding a union of all pairs generated by all independent passes, plus all those pairs that can be inferred by transitivity of equality (on record ids).

An annotation for MPN within the matching of Example 4.1 could be the following: %MPN: key = title; window size = 100%. Note that only the parameters of the physical algorithm are specified in the absence of particular mathematical properties of the algorithm. If the previous annotation was also given, the system would have the choice between three possible implementations of the matching operator.

5 Experiments

The purpose of our experiments is to assess the value of our proposed framework. More specifically, we evaluate the effectiveness of our declarative language for expressing a data cleaning strategy at a logical level (qualitative analysis), and the possibility of orientating the selection of alternative implementations of matching using annotations in the logical specification of matching (quantitative analysis).

We ran our experiments on a single-CPU Pentium III workstation with a i686 CPU at 501MHz, cache size 512KB, and 1G bytes of RAM having Linux as its operating system. We used ORACLE8i as our database system and JDK1.3. to execute the Java code.

5.1 Qualitative Analysis

The data cleaning strategy used in the Citeseer example follows the general principles of previously published data cleaning methodologies (e.g., [9]). The first step for modeling the data cleaning application for bibliographic references was to determine the flow of transformations for cleaning the data. The high level nature of the language and the small number of available operators enabled a rapid prototyping of the graph of operations[14]. The initial strategy was designed and tried with less than one hundred records taken from the Citeseer dirty data set. The graph of operations remained stable after the initial stage of specification of the data cleaning strategy[15] although the specification of the content of each operator, including the design of the external functions, evolved considerably over time as the amount of dirty data considered increased.

[14]This was corroborated by previous experiments with the database of INRIA suppliers and information concerning clients of an European Telecommunication company.

[15]From the 100 operations that compose the data cleaning program, around 90% were designed from the beginning.

Having separate logical operations for matching, clustering and merging is quite important because these operations are logically independent (see e.g., [1]) and hence must be specified separately. On the contrary, in some proposals such as [13] and [22], a single algorithm is provided to implement both a matching using the MPN method and a clustering using transitive closure. However, although the usage of the optimization for matching may be worthwhile in terms of the ratio of execution time to accuracy, the use of the transitive closure may be inappropriate (e.g., for clustering duplicate author names). Controlling the separate usage of these operations is thus important.

We observed how refinement was important for finding the right matching criteria for author names, event names and publications. In fact, the criteria that lead to maximum quality highly depends on the domain handled. In our case, the criteria for matching author names are far more complex than the criteria for comparing titles, for instance. Several heuristics (e.g. "Dallan Quass" and "D Quass" are considered very close although the Damerau-Levenshtein edit distance returns a distance of 5) were applied in order to find the best accuracy for author matches. Nevertheless, the matching methods did not consider any knowledge of the domains (e.g., a dictionary of US names). This was particularly difficult in this application due to the heterogeneity of names found. To illustrate the effectiveness of the approach followed, we manually cleaned a subset of 1,000 dirty bibliographic references[16] and compared the corresponding clean author names with the results of applying the data cleaning program to the same dirty data set. We observed a recall of 0.7 and a precision of 0.92, where recall gives the fraction of correct authors (considering the manual cleaning returns all the correct authors) found by the data cleaning program and precision is the fraction of authors automatically found that are indeed correct authors.

The generation of exceptions and the data lineage mechanism provided by our system were helpful in refining the cleaning criteria.

5.1.1 Exceptions, User Interaction and Data Lineage

The design and refinement of the data cleaning program for Citeseer provided an assessment of the data lineage mechanism supported by our framework. We illustrate the value of the exception and data lineage (aka data provenance or data pedigree in [2]) mechanism by means of the merging operation presented in Example 3.12. In this example, the function getLongestAuthorName() throws an exception if there is more than one author name with maximum length. In that case, a tuple is written in the exceptional output data flow named $\mathsf{MergeAuthors}^{exc}$. Suppose the following tuple is inserted in the exception relation output by the MergeAuthors operation:

$\text{MergeAuthors}^{exc}$: 2 | EqualSizeException

[16] The subset of data was chosen to include author names with different lengths and origins; different ways of writing the same event; errors in the title and inconsistencies in data.

The user can then trace back this tuple to the input tuples of ClusterAuthors and DirtyAuthors that have generated it, by soliciting explanations from the corresponding merging and clustering operations.

ClusterAuthors: 2 | 2, 10

DirtyAuthors: 2 | A Gupta
10 | H Gupta

The explanation of this exception allows the user to discover that his interaction is needed because the system failed to choose an author name and that she has to take a corrective action. The user may insert directly the correct author name into the Authors relation, if "A Gupta" and "H Gupta" are indeed the same person. Otherwise, the user may update the corresponding DirtyAuthors tuples so that they are no longer considered as candidate matches by the matching operator (e.g., expand to "Ashish Gupta" and "Himanshu Gupta").

As the size of samples of bibliographic references increased, we observed two frequent uses of the explaining mechanism: one led the user to perform manual cleaning (as in the above example) to disambiguate a situation that could not be automated, while the other led the user to investigate exceptions created as a result of an incorrect cleaning criteria implemented by external functions. The best examples of the second case are the extraction operations (e.g., the Extraction mapping presented in Section 3 that extracts titles, names of authors and events from a textual bibliographic reference). The very unstructured nature of the initial data set required a stepwise refinement of the external functions embedded in the mapping operator. For instance, a first criteria of extraction threw an exception for the following tuple:

16660|Th. Back, D. B. Fogel, and Z. Michalewicz, editors. Handbook of Evolutionary Computation. Oxford University Press, New York, and Institute of Physics Publishing, Bristol, 1997

16660|CiteSeerException - AuthorListCouldNotBeExtracted

This happened because "Th." is a name abbreviation and the initial extraction criteria included a heuristic stating that a "." after a word that does not belong to the dictionary of abbreviations is a separator. Hence, the recognition of authors stopped after "Th". The problem is solved by adding "Th." to the list of possible abbreviations. After modifying the logic of the corresponding external function, called within the extraction mapping, the correct tuple was returned as output of the extraction:

16660|Th Back, DB Fogel, Z Michalewicz|Handbook of Evolutionary Computation| Oxford University Press , and Institute of Physics Publishing

As a final note, the refinement of criteria for extraction eliminated some of the exceptions, but not all of them. In the cleaning program for bibliographic references, the number

of exceptions thrown during extraction of authors, title and event names is still 10% of the number of the initial dirty tuples. This led us to conclude that an environment for building extraction functions (e.g., using learning techniques) should be used; otherwise manual extraction is heavily required[17] .

5.2 Quantitative Analysis

We now report on the usefulness of separating the logical specification of a matching operation from its physical implementation.

We compared the performance of three distinct physical algorithms for the matching operator: nested loop (NL), that corresponds to the naive semantics of the matching; multi-pass neighborhood method (MPN) and neighborhood join (NJ). The last two were presented in section 4 and correspond to optimizations of the first algorithm. All algorithms were implemented in Java.[18]. Since the MPN method computes an approximation of the candidate matches defined by a matching operation, we used a measure of quality, called *recall*, defined as the number of matches returned by the algorithm divided by the number of candidate matches.

We applied these algorithms to the matching of authors, events, and publications for two subsets of dirty bibliographic references with respective cardinalities of 100,000 records (15MB), and 500,000 records (86MB). The matching of authors (resp. events) computes the pairs of duplicate author names (resp. event names), while the matching of publications considers two publications as duplicates if their titles are close enough and the corresponding events are equal (this is an extension of the MatchDirtyTitles matching presented in example 4.1). All matching operations use the same distance criteria (based on the Damerau-Levenshtein distance).

Experiment 1: Tuning the MPN method The goal of this experiment is to measure the trade-off between the execution time and the quality of the data in the case of the MPN method.

The MPN method has three parameters: a sort key, a window size and the number of passes of the algorithm[19]. The sort keys tested were the following: LTR and RTL - sort entries by author name and alphabetic order from left to right and from right to left, respectively; S - sorts entries by author name without blank spaces; and L - sort entries by author's last name. We combined these keys in order to run the algorithm for 1, 2 and 4 passes. The window sizes tested were: 100; 1,000; and 10,000.

[17]Note that the difficulty found in extracting accurate data was due to the strong unstructured nature of our data set, which is not generalized to all situations.

[18]A fourth physical algorithm where the optimization capabilities of an RDBMS could be used to execute the matching operation was also considered. It was not taken into account in the measurements since most RDBMS do not optimize approximate joins as already mentioned.

[19]The algorithm includes a step where a transitive closure clustering algorithm is applied to group duplicates together. In our experiments, we computed only the matching phase (sorting and window-scanning).

We varied the three parameters in order to *(i)* measure the minimum execution time for a recall superior to 0.8, and *(ii)* obtain a recall as close to 1 as possible. Table 8 gives the results obtained for 500,000 dirty entries (that correspond to 133,000 distinct dirty authors). The best execution time for a recall superior to 0.8 and the highest recall value are in bold. As expected, we observed that a large variation of the execution times (from 4.9 to 745 min) does not correspond to a large range of recall values (0.806 to 0.987). In fact, in a short period of time, a reasonable recall is obtained (20.7 min/0.91). From this point, recall increases very slightly whereas the execution time increases drastically.

Window size	Sort Key:Nb Passes	Time(min)	Recall
100	LTR+RTL:2	**4.9**	**0.806**
100	LTR+RTL+S+L:4	8.5	0.814
1000	LTR:1	19.8	0.68
1000	RTL:1	20.7	0.91
1000	LTR+RTL:2	40.1	0.927
1000	LTR+RTL+S+L:4	74.4	0.937
10,000	RTL:1	199.6	0.957
10,000	LTR+RTL+S+L:4	**745.0**	**0.987**

Table 8: Execution of the MPN method for matching 133,000 dirty authors

Experiment 2: Comparing MPN and NJ executions In this experiment, we compared the results of MPN and NJ. The NJ used the distance filtering presented in section 4 that is a distance filter: map=length; dist=abs[20]. By definition, the NJ algorithm always has a recall of 1.

Table 9 shows the results obtained[21]. The columns named MPN refer to the multi-pass neighborhood algorithm and the last column, named NJ refers to the neighborhood algorithm. Let us pick the fourth line of the table that corresponds to the results presented in table 8. The first MPN(a) column contains the minimum execution time (T = 4.9) to obtain a recall (R = 0.806) superior to 0.8. The second MPN(a) column registers the window size (WS = 100) and the number of passes (NP = 2) needed to obtain such time and recall values. The two MPN(b) columns report the same measures (T = 745.0, R = 0.987) and parameters (WS = 10,000; NP = 4) for obtaining a maximum recall value. The last column represents the execution time (T = 790.0) of the NJ algorithm that returns a recall equal to 1

Comparing the time/recall columns, we conclude that in general, MPN can be faster but less accurate than NJ. If execution time is crucial and accuracy can be neglected to a

[20]The selectivity of the filter (percentage of comparisons computed) was: 11% for events (filter based on the length of the event name); 30% for authors (filter based on the length of the last name and on the length of the entire name without blank spaces); and 16% for publications (filter based on the length of the title).

[21]The execution times obtained for the NL execution (for 100,000 tuples) were 30, 180 and 4 minutes for Authors, Events and Publications, respectively. The execution times obtained using the NJ (as shown in table 9) for the same subsets of data correspond to gains of 33%, 72%, and 68% respectively.

Dirty tuples	Matching	MPN(a):T/R	MPN(a):WS/NP	MPN(b):T/R	MPN(b):WS/NP	NJ:T/R
100,000	Authors	0.4/0.915	100/1	58.95/0.998	10,000/2	25.8
100,000	Events	6.9/0.846	100/2	398.1/0.999	10,000/2	66.0
100,000	Publications	0.17/0.887	100/1	6.27/0.999	10,000/2	1.7
500,000	Authors	4.9/0.806	100/2	745.0/0.987	10,000/4	790.0
500,000	Events	454.5/0.862	1,000/2	3607.0/0.975	10,000/2	2555.0
500,000	Publications	0.73/0.861	100/1	77.0/0.996	10,000/2	63.0

Table 9: Implementation of the matching using MPN and NJ. T/R is the execution time/recall ratio. WS is the window size, and NP is the number of passes for the MPN method.

certain level, MPN is worthwhile; otherwise NJ is the best choice. The differences observed depend on the domain of data the matching is applied to. For event names, whose values are strongly unstructured (even if a normalization against a dictionary has been applied), the NJ algorithm is able to achieve a recall of 1 much faster than the MPN method; this difference is less remarkable for author names. Thus, the results of this experiment confirm the usefulness of providing more than one physical implementation for the same matching operator.

6 Related Work

Related work falls into three main categories: high level languages to express data transformations, data cleaning frameworks, and algorithms to support matching, clustering and merging operations.

Several languages have been recently proposed to express data transformations: SQL99 [12], WHIRL's SQL [5, 6], and SchemaSQL [19]. Our language supports operations such as clustering and merging that are not expressible in SQL99. Furthermore, in SQL the occurrence of an exception immediately stops the execution of a query. In contrast, our semantics enables to compute the entire set of tuples that caused the occurrence of exceptions. Last, unlike SQL [21], our language enables the optimization of a matching operation by making it a first-citizen operator in the language. WHIRL's SQL extends SQL queries with a special join operator that uses a similarity comparison function based on the vector-space model commonly adopted in statistical information retrieval. This join operator is a special case of our matching operator. Furthermore, WHIRL does not support clustering and merging operations. SchemaSQL is a powerful extension of SQL that includes operations to restructure a relational schema. This language is useful to perform integration queries in relational multi-database systems. It is complementary to our language in the sense that our language does not allow schema restructuring operations whereas SchemaSQL does not allow neither arbitrary clustering and merging operations nor optimized matching.

Several frameworks have been proposed for data integration and cleaning. We already compared our work to commercial ETL and data cleaning tools. Research prototypes include

[20] and Potter's Wheel-ABC [24]. Lee et al [20] propose a rule-based approach to express matching, clustering and merging operations, which is implemented using the Java Expert System Shell. However, their framework provides a fixed matching algorithm (the MPN method) with certainty factors attached to the matching rules in order to minimize the degradation of precision due to the application of transitive closure in MPN. Also, the approach does not clearly scale up for very large data sets due to the use of an expert system shell. Like us, Potter's Wheel promotes an interactive approach whereby users are able to apply a set of simple transformations to modify samples of data and see the results interactively. However, unlike us, their focus is on interleaving data transformation and discrepancy detection to facilitate the refinement of data transformations. Their techniques for automatic discrepancy detection that run as a background process behind the data transformation could be applied to our context. Other frameworks have been proposed for data integration such as the Squirrel system [29, 30] and the DWQ integration method of [3]. However, the data cleaning aspects of data integration are marginally addressed by these research projects.

Finally, several algorithms have been proposed to implement matching, clustering, and merging operations. [13] presents the MPN algorithm described earlier that has been further optimized in [22] using a tighter integration of the matching and clustering phases.

7 Conclusions

We presented a data cleaning framework whose main originality is a separation in two clear layers: logical and physical. The main features of the framework described in this paper are: *(i)* a declarative language to specify the flow of logical transformations; *(ii)* a declarative specification of user interaction based on the automatic generation of exceptions during operator's execution, and *(iii)* a declarative way to select an optimized implementation for the matching operator. Our experiments assessed the value of this framework both qualitatively and quantitatively using a large data cleaning program on a real data set of bibliographic references. In particular, we showed that the separation between the logical specification of a matching and its implementation gives a proper control to the user of the tradeoff that arises between performance and recall. The claimed advantages of our framework have been recently acknowledged by [7], which describes an industrial prototype for XML data integration that implements a data cleaning framework directly inspired by ours.

References

[1] T. R. Belin and D. B. Rubin. A method for calibrating false-match rates in record linkage. *Journal American Statistical Association*, 90(430), June 1995.

[2] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, 2001.

[3] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. A principled approach to data integration and reconciliation in data warehousing. Workshop on Design and Management of Data Warehouses (DMDW'99), 1999.

[4] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, March 1997.

[5] W. Cohen. Integration of Heterogeneous Databases without Common Domains Using Queries based on Textual Similarity. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1998.

[6] W. Cohen. Some practical observations on integration of Web information. In *WebDB'99 Workshop in conj. with ACM SIGMOD*, 1999.

[7] D. Draper, A. Halevy, and D. S. Weld. A Commercial XML Data Integration System: Lessons Learned. In *Proc. of ACM SIGMOD Conf. on Data Management*, 2001.

[8] EDD. Home page of DataCleanser DataBlade Module. http://www.npsa.com/edd/.

[9] L. P. English. *Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits.* Wiley, 1999.

[10] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equit. Efficient and effective querying by image content. *JIIS*, 3(3/4), 1994.

[11] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An Extensible Data Cleaning Tool. In *SIGMOD (demonstration paper)*, 2000.

[12] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* R&D Books, 1999.

[13] M. A. Hernandez and S. J. Stolfo. The Merge/Purge problem for large databases. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1995.

[14] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data Cleansing and the Merge/Purge problem. *Journal of Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[15] Informatica. Informatica home page. http://www.informatica.com/.

[16] N. R. Institute. Research Index (CiteSeer). http://citeseer.nj.nec.com/.

[17] E. T. International. Home Page of ETI. http://www.evtech.com.

[18] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall Advanced Reference Series, 1988.

[19] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *Proc. of the Int. Conf. on Very Large Databases*, Mumbai, India, 1999.

[20] M. L. Lee, T. W. Ling, and W. L. Low. A Knowledge-Based Framework for Intelligent Data Cleaning. *Information Systems Journal - Special Issue on Data Extraction and Cleaning*, 2001.

[21] J. Melton. *(ISO-ANSI Working Draft) Persistent Stored Modules (PL/PSM)*. ISO, 1999.

[22] A. Monge. Matching Algorithms within a Duplicate Detection System. *IEEE Data Engineering Bulletin*, 23(4), December 2000.

[23] E. Rahm and H. H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(3), September 2000.

[24] V. Raman and J. M. Hellerstein. An Interactive Framework for Data Transformation & Cleaning. Working draft: http://control.CS.Berkeley.EDU/abc/.

[25] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, 2000.

[26] Sagent. Sagent home page. http://www.sagenttech.com/.

[27] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Theory*, 147:195–197, 1981.

[28] Vality. Home page of the Integrity tool. http://www.vality.com/html/prod-int.html.

[29] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.

[30] G. Zhou, R. Hull, R. King, and J.-C. Franchitti. Using Object Matching and Materialization to Integrate Heterogeneous Databases. In *COOPIS*, 1995.

Appendix 1 - Formal Definition of the Mapping Operator

Definition 7.1: (Valuation of a functional expression in let-clause). Let E be a let-clause wrt $S\{A_1, ..., A_n\}$ that defines $P_1, ..., P_q$. Let $P_i = f_i(\vec{x}, \vec{y_i})$, $1 \leq i \leq q$ be an assignment statement in E, where $\vec{y_i} = y_i^1, ..., y_i^k$. A *valuation* ν of $f_i(\vec{x}, \vec{y_i})$ wrt a fact $S(\vec{a})$ in a database instance I, is a mapping recursively defined as follows:

1. $\nu(f_i(\vec{x}, \vec{y_i})) = f_i(\vec{a}, \nu(y_i^1), ..., \nu(y_i^k))$ where $S(\vec{a})$ is a fact over S in I
2. if $y = P_j$, for some j, $1 \leq j \leq q$, and P_j is defined by an atomic assignement statement, then if there exists a fact $P_j(\vec{a}, \vec{b})$ in I, then $\nu(y) = \vec{b}$; otherwise $\nu(y)$ is undefined.
3. if $y = P_j$, for some j, $1 \leq j \leq q$, and P_j is defined in a table assignement statement, then if the set of all facts $P_j(\vec{a}, \vec{b})$ in I is not empty, then $\nu(y)$ is equal to this set of facts; otherwise $\nu(y)$ is undefined.
4. if y is a variable ranging over the domain of an attribute B of a predicate P_j, and P_j is defined by an atomic assignment statement, then if there exists a fact $P_j(\vec{a}, .., b, ...)$ in I where b is a value for attribute B, then $\nu(y) = b$; otherwise $\nu(y)$ is undefined.
5. if y is a variable ranging over the domain of an attribute B of a predicate P_j, and P_j is defined by a table assignment statement, then if the set of all values b such that $P_j(\vec{a}, ..., b, ...)$ is a fact in I, where b is a value for attribute B, is not empty, then $\nu(y)$ is equal to this set; otherwise $\nu(y)$ is undefined.
6. if y is a constant a, then $\nu(y) = a$.
7. if y is a functional expression $g(t_1, ..., t_m)$ then $\nu(g(t_1, ..., t_m) = g(\nu(t_1), ..., \nu(t_m))$
8. $f_i(\vec{a}, \nu(y_i^1), ..., \nu(y_i^k))$ is undefined if there exists j, $1 \leq j \leq k$, such that $\nu(y_i^j)$ is undefined.

Definition 7.2: (Semantics of a let-clause). Let E be a let-clause wrt S that defines $P_1, ..., P_q$. The semantics of E is recursively defined by a function, noted $\mathsf{Let}(E, I)$, that maps a database instance I into a new database instance J as follows[22] .

1. If $E =< e_1 > \bullet E_2$, where E_2 is the tail of E, then $\mathsf{Let}(< e_1 > \bullet E_2, I) = \mathsf{Let}(E_2, \mathsf{Let}(< e_1 >, I))$
2. If $< e >$ is an atomic assignment statement of the form $P_i = f_i(\vec{x}, \vec{y_i})$. Then, $\mathsf{Let}(< e >, I)$ is such that:
   - if $S(\vec{a})$ is a fact in I, and ν is a valuation of $f_i(\vec{x}, \vec{y_i})$ wrt $S(\vec{a})$ such that ν is defined, $b = \nu(f_i(\vec{x}, \vec{y_i}))$, and $b \neq \mathsf{exc}$ then $P_i(\vec{a}, b)$ is a fact in J

[22] We shall use the following notations for lists: $\bullet$ denotes the append operator and list constructor for lists and $< e >$ denotes the atomic list with an element e.

- if $S(\vec{a})$ is a fact in I, and one of the two following conditions hold: (i) $b = \nu(f_i(\vec{x}, \vec{y_i}))$ and $b = \mathsf{exc}$ or (ii) $\nu(f_i(\vec{x}, \vec{y_i}))$ is undefined, then $S^{exc}(\vec{a})$ is a fact in J

3. If $< e >$ is table assignment statement of the form $P_i = f_i(\vec{x}, \vec{y_i})$, then $\mathsf{Let}(< e >, I)$ is such that:

   - if $S(\vec{a})$ is a fact in I, and ν is a valuation of $f_i(\vec{x}, \vec{y_i})$ wrt $S(\vec{a})$ such that ν is defined, and $\nu(f_i(\vec{x}, \vec{y_i}))$ contains a fact $(\vec{a}, \vec{b})$ then $P_i(\vec{a}, \vec{b})$ is a fact in J .
   - if $S(\vec{a})$ is a fact in I, and one of the two following conditions hold: (i) $\nu(f_i(\vec{x}, \vec{y_i})) = \mathsf{EXC}$ or (ii) $\nu(f_i(\vec{x}, \vec{y_i}))$ is undefined, then $S^{exc}(\vec{a})$ is a fact in J .

Definition 7.3: (Semantics of a filter). Let Φ be a filter over $P_1, ..., P_q$. The semantics of Φ defines a function, noted $\mathsf{filter}(\Phi, I)$, which maps a database instance I into an instance J such that there exists a predicate U such that $J[U]$ is the set of facts $(a_1, ..., a_m)$ of $I[P_1] \times ... \times I[P_q]$ for which $\Phi(a_1, ..., a_m)$ evaluates to true.

Definition 7.4: (Semantics of a constraint-clause). Let Σ be a constraint-clause over $P\{B_1, ..., B_m\}$ wrt $\{A_1, ..., A_n\}$. The semantics of the constraint-clause is defined by a function, noted $\mathsf{check}(\Sigma, I)$, that maps an instance I into J as follows:

- there exists a predicate P^{ok} such that if $P(b_1, ..., b_m)$ is in I, and for each σ of Σ, $\sigma(b_1, ..., b_m)$ evaluates to true then $P^{ok}(b_1, ..., b_m)$ is in J
- if $P(b_1, ..., b_m)$ is in I, and there exists a constraint σ of Σ such that $\sigma(b_1, ..., b_m)$ evaluates to false then $\mathsf{project}(P\{A_1, ..., A_n\}, P(b_1, ..., b_m))$ is a fact over S^{exc} in J .

Definition 7.5: (Semantics of the mapping operator). A mapping operator $(S, E, \Phi, \mathcal{T}, \mathcal{C})$ defines a function, noted MAP, that maps a database instance I into an instance J as follows. For each i, $1 \leq i \leq q$, let $pre\_T_i\{A_1, ..., A_n, B_{i1}, ..., B_{ik}\}$ be a relation associated with $T_i\{B_{i1}, ..., B_{ik}\}$. Then, we have:

- for each i, $1 \leq i \leq q$, $J[pre\_T_i] = \mathsf{project}(U\{A_1, ..., A_n, B_{i1}, ..., B_{ik}\}, \mathsf{filter}(\Phi, \mathsf{Let}(E, I)))$, where U is the predicate defined by the filter Φ
- for each i, $1 \leq i \leq q$, $J[T_i] = \mathsf{project}(pre\_T_i\{B_{i1}, ..., B_{ik}\}, \mathsf{check}(\Sigma_i, J[pre\_T_i]))$, where Σ_i is the constraint clause over U wrt S containing all the conditions of $\mathcal{C}_i$
- if $S^{exc}(\vec{a})$ is in $\mathsf{Let}(E, I)$, or $S^{exc}(\vec{a})$ is in $\mathsf{check}(\Sigma_i, J[pre\_T_i])$ then $S^{exc}(\vec{a})$ is in J.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399